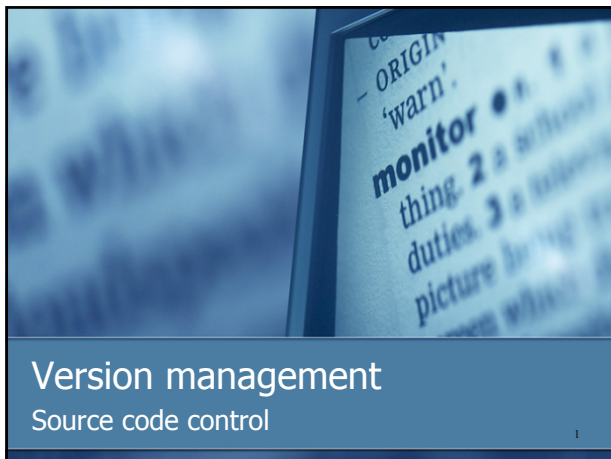




Version management

Source code control

1

"Program versions" (releases)

- In a commercial/industrial setting, there will typically be several versions of a given program:
 - version 1.0, still used by some customers
 - version 1.1-WVista, special modified version to suit Windows-Vista
 - (versions 1.2 ... 1.4; experimental things created while working toward version 1.5)
 - version 1.5, the version currently being sold
 - version 1.6, an experimental version being developed by group A
 - version 1.7, an experimental version being developed by group B
- Versions will share different subsets of common files and each have their own unique files.

4

Programs ...

- Programs written at Universities are "special"
- **The undergrad's program:**
 - Implemented in haste
 - Untested, unchecked
 - "Passes" if runs once on a set of test data
 - Discarded.
- **The graduate student's program:**
 - "It is 90% done"
- **The researcher's program:**
 - "I have this superb idea - but there's no need to write a program to test it. The idea is so simply superb. It will go down fine at that conference in Tuscany; I've always wanted a holiday in Tuscany."

2

Program Versions

- Why do you need to keep them?
- Scenario 1: a customer with Version 1.0 reports a bug that hasn't been noticed previously – you'll need to work with the 1.0 code to track down the bug, this will allow you to send out a "patch" (you will also have to check whether the bug affects later versions).
- Scenario 2: the new ideas proposed by groups A and B are both great – you'd like to merge them to form version 1.8

5

Programs ...

- Programs in a commercial/industrial setting are different.
- Developed as a product
 - Developed by several people working together
 - "Sold" (if only to another part of same organisation)
 - Maintained
 - Evolved into "new, improved Mark II" product
- The maintenance and evolution processes become complex and require support tools.

3

Program versions

- Note: it isn't just your own code that you may have to keep.
- Of course, you have to keep the source versions of the design documents, the user-manual, the help files, ...
- But it isn't only your stuff that you may have to keep
 - Version 1.1WinVista was compiled with the XYZ compiler (no longer supported); you'd better have a copy of the XYZ compiler and all its associated libraries just in case you need to make a patch for a bug in the 1.1WinVista version.

6

Program versions

- You could simply keep complete copies of each version (and all required system software).
- Often too costly in terms of storage.
- Also
 - doesn't give you any model for understanding the changes made between versions (they are just similar programs)
 - doesn't let you trace reasons for changes
 - doesn't help you do things like combine versions
- So total backup of versions is inadequate, you need a more sophisticated system.

7

This overview

- Will start with what is really a historical perspective
 - the material comes in part from the Bolinger & Bronson book "Applying RCS and SCCS"
- Then look at Subversion
 - some illustrations are taken from the "subversion" book

10

Version/Configuration management tools

- Basic idea:
 - Control the development process
 - Let several developers work simultaneously
 - Have a mechanism that prevents you from getting into a mess by having two people changing different copies of the same code
 - Either you disallow this possibility
 - Or you have some mechanism for resolving issue of multiple copies of code with varied changes
 - Keep track of what was changed (and why) between versions
 - Allow reconstruction of any chosen version
- (*Configuration management* — tends to imply building versions for different computing systems. So *version management* is generally the preferred term.)

8



Conventional version management SCCS inspired example

11

The tools

- Examples
 - Source Code Control System (classic Unix, SystemV Unix)
 - Revision Control System (Berkeley Unix, Linux)
 - Microsoft's proprietary systems
 - "Code Versioning System" (CVS)
 - Subversion
- The tools share some concepts and provide generally similar services – but do use different terminology and adopt different approaches to tracking changes and controlling development work.

And now there is "git" – a very different creature

9

SCCS? Why bother?

- SCCS has worked for more than quarter of a century!
- Still there (either SCCS or RCS which isn't too different)
 - SCCS is on banshee
 - Some of the Linux machines have RCS installed
- Can use SCCS to maintain versions for assignments in other CS subjects for which there is no subversion server.
- Worth playing with SCCS a little
 - Just to see the basic mechanisms at work
 - You can look inside SCCS archive files and get quite a detailed view of its operation!

Read the "man" pages for more details of use of SCCS/RCS if you do decide to use them.

12

SCCS and similar

- Code versioning evolved on time-share systems – like the original Unix.
- Created for groups of people working collaboratively on the development of a project.
- Made use of common file-space that could be accessed by all users – (think of shared directories like /share/cs-pub on the main banshee system)

13

What does SCCS do? Start-up

1. **Create the initial archive**
 - An initial set of files for the project could be deposited in the archive.
2. **Individual group members (developers) could then "check out" files that they want**
 - Some files "checked out" as read-only
 - Developer needs some code already implemented
 - Some files "checked out" for modification
 - Developer wants to correct or extend some code.
 - (When more than one version of a file exists, the developer can specify which version they want to "check out")

16

The shared code archive

- The "official" project code would be kept in a group directory accessible by all members of the development group.
- The "archive" will just be a subdirectory, each file in the project is managed separately – so each exists as a special "sccs archive file" in this subdirectory.
 - If you are using SCCS for your own CS assignments you would just use a subdirectory in your home directory
- Each developer would have copies of the code that they were actively modifying.

14

SCCS example - starting

- You have a file `coms.cc` which is the initial version of your program.
- Create the archive from this initial source version (here, will get created in current directory):

```
$ admin -icom.s.cc s.coms.cc
```

- SCCS archive files get names starting `s.`
- After creating the archive you would
 - remove the existing `coms.cc` file
 - check out, "get", a read-only copy (or a modifiable copy)
 - (if you don't do this, your existing read/write original is going to confuse subsequent operations)

17

A command-line "source code control" application set

- Code in the shared "archive" would be accessed only via the various small programs that made up the "source code control system"
- Typical "Unix guru" style
 - Apparently unrelated command-line commands
 - Each command taking a bizarre collection of command line arguments

15

SCCS example - checkout

- Checking out a read-only file

```
$ get s.coms.cc
```

- This will get the most recent revision of `coms.cc` (if the request is successful, the number of lines in the file is printed).
- You can request a specific revision:

```
$ get -r1.2 s.coms.cc
```

18

SCCS example - checkout

- Checking out a modifiable revision:

```
$ get -e s.coms.cc
```

- Gets the (most recent) revision, e.g. 1.3.
- Assumes that you want to make the next revision (implicitly 1.4)
- If command successful, should get response like

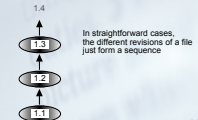
```
1.3  
New delta 1.4  
635 lines
```

19

What does SCCS do? Update

3. Individual group member (developers) "checks in" an updated file

- System has to:
 - Identify what has been changed in this update.
 - Create a file that has both original code and the update
 - Subsequently, may want to retrieve either the most updated version of the file – or one of the earlier versions.
 - The system has to record sufficient information to allow this to be done.
- Record "revision" number.



22

SCCS example

- Checking out a modifiable version.
- If another group member already has a modifiable version, the archive file will contain a "lock" note.
- In this case, you should get a response like:
Error s.coms.cc being edited
'1.4 tg015 10/4/2008'
- response should give user identifier of person (tg015) already editing the revision you requested.

20

SCCS example – "delta"

- When you have finished making (and testing) changes, you can put your file back:

```
$ delta s.coms.cc
```

- The delta command will prompt for comments:
comments? Added provision for timeout\
on connect request.
- The delta command should then produce a report summarizing the extent of changes:
20 inserted
5 deleted
699 unchanged

23

File permissions set by SCCS

- File access modes:
 - Unix files can be "readable, writeable, executable"
 - access is allowed at three levels – owner, group, other
 - mode examples -rw-rw-r--, -rw-r-----, -r--r-----
 - a file "checked out" for modification would be -rw- for owner (and possibly no other access)
 - a file "checked out" for reading would be -r--
 - *don't change the modes, you'll disrupt the working of the source code management tool*

21

SCCS delta

- The delta command deletes your copy of the file once it has been returned to the archive.
- You are expected to get a read only copy once you have finished with a set of changes.
- If you got a file for modification, but decide not to make changes, you have to use the unget command to clear the lock that you hold in the archive file:

```
$ unget s.coms.cc
```

24

Revision number

- Revision number
 - multipart number e.g. 1.7, 2.4, 3. 6
 - major revision number (1, 2, 3)
 - minor revision number (.7, .4, .6)
 - (and additional fields when have branched development "trees")

25

SCCS branching

- Creating a branch:

Current revision > 1.2

```
$ get -e -r1.2 s.thing.cc
```

1.2

new delta 1.2.1.1



28

SCCS Major revision number

- You can change the revision major number when using `get` to obtain a modifiable file.
- Suppose current revision is 1.7, then


```
$ get -r2 s.coms.cc
```
- would produce a response like


```
$ get -r2 s.coms.cc
```

1.7

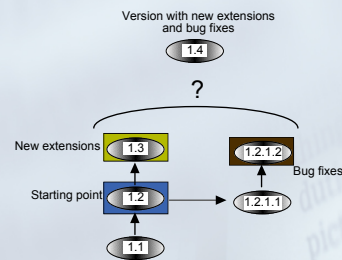
new delta 2.1

704 lines

26

What does SCCS do? Merge

5. Sometimes you want to "merge" code from different branches:



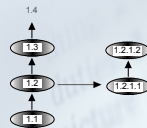
29

What does SCCS do? "Branch"

4. Typically, there is a linear progression through versions
 - Version-1.1, Version-1.2 etc

Sometimes you need to create branches

 - An experiment?
 - Try out a different algorithm
 - A special requirement?
 - Two target platforms with different versions of library functions; need slightly different versions of code invoking the differing underlying libraries.



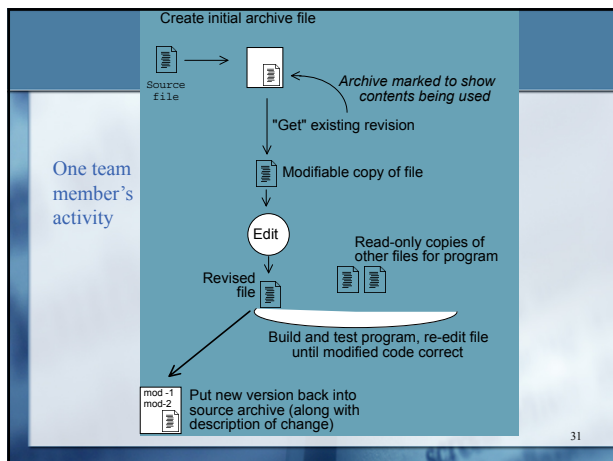
Branching tree structured revision hierarchies do occur

27

"Three way merges"

- Have common "ancestor" file (revision 1.2)
- Have two "descendant" files with different changes.
- Want a new "descendant" file with combined changes.
- With SCCS this isn't fully automated!
 - Unix utility program `diff3` provided to help process.
 - `diff3` produces a set of editing commands for use in `ed`.

30



SCCS Archive files

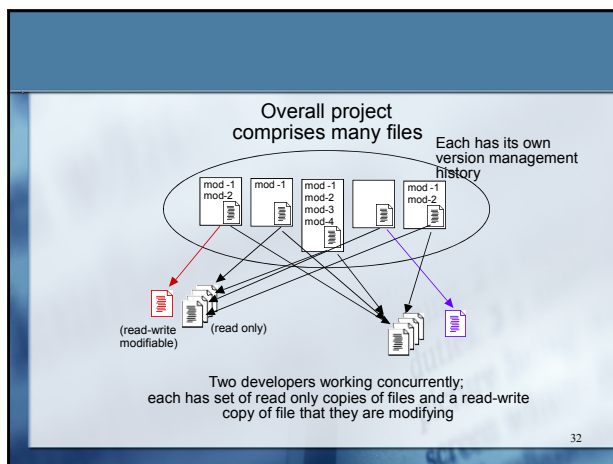
- The "archive" files don't contain a complete copy of each successive version of the corresponding source file.
- Instead:
 - one copy of the source text
 - sets of editing commands that can be used by a "script editor" (e.g. sed) to generate any required version.

```

mod-1
date
reason
author
10i
#include <fstream.h>

```

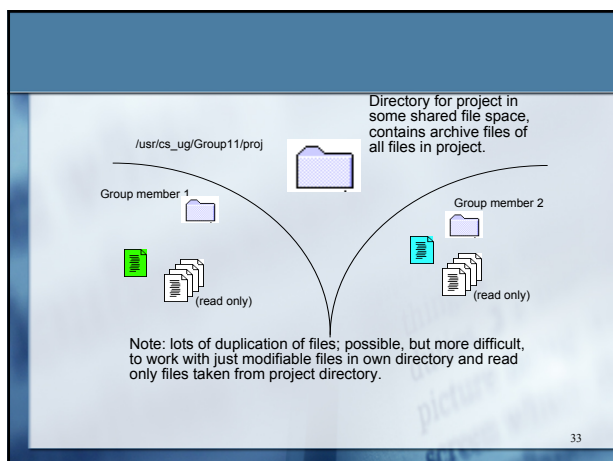
34



SCCS Archive files

- SCCS:
 - store the original version as used to create the archive
 - store edits corresponding to each change
- so if you want to retrieve fifth version (version 1.5)
 - SCCS gets the original text out (version 1.0)
 - SCCS runs "sed" to do editing for the changes needed to get version 1.1
 - SCCS then feeds "sed" with edits to bring file up to version 1.2, then to 1.3, 1.4, finally 1.5
- ("checking out" a file can take a noticeable amount of time)

35



SCCS Archive files

- File "check in"
 - this also a moderately elaborate process
- For example, suppose you checked out version 1.5, edited it and wanted the new file to be version 1.6
- SCCS would have to
 - create a file with the 1.5 version for comparison purposes
 - use diff to compare your modified version with the 1.5 version
 - diff reports differences by showing the ed/sed editing commands needed to change old version to new version
 - save these editing commands along with explanatory comments that you provide

36

SCCS/RCS Archive files

- SCCS keeps original and the edits needed to build subsequent versions.
- RCS keeps most recent, and the edits to get back to earlier versions (well, something like that).
- If you decide that you will never want certain intermediate versions (e.g. no one received copies of versions 1.3 and 1.4) you can tell SCCS/RCS to eliminate these – you'll be left with a single set of editing commands for going between version 1.2 and version 1.5.

37

SCCS identifier keywords.

- SCCS of course designed to work with C program code.
- It provides a mechanism for embedding the version number, modification date, etc into the code that you get when you extract a file.
- SCCS uses convention %<letter>%
 - different %<letter>% combinations have different defined expansions
 - %E% becomes the date
 - %M% "module name" (by default, file name)
 - %I% revision number

40

SCCS: Files and programs

- Note these tools work with versions of the constituent *files*, not the complete program.
- Version 1.5 of the program might incorporate:
 - main.cc version 1.2
 - collections.cc version 1.8
 - coms.cc version 1.4
 - ...
- The simpler older tools (like SCCS) don't provide much help in keeping track of such relations, they focus solely on the individual components (files) and their histories.

38

SCCS labelling strings

- Use these to initialize static (filescope) character strings (allows identifier information to be included in both source and compiled versions, also info. can be printed out in debug or other reports).
- e.g. original file has:

```
static const char SCS_ID[ ] =  
    "%Z% %M% %I% %E%";
```

- expanded version when you get the file

```
static const char SCS_ID[ ] =  
    "@(#) coms.cc 2.2 07/06/01";
```

41

Version of a program ...

- How to get "program version management" with a file management tool?
- You have a text file in the set of archived files, the contents of this file describe program versions (e.g. by listing the appropriate file versions, compilers, libraries that are needed to create each program version).
- You extract the text file version corresponding to the program version to get configuration details.
- You manually extract the specified file versions (without automation, it is a bit tedious and error prone).

39

SCCS labels

- The @(#) string produced from %Z% token is used by SCCS utilities to flag lines with revision information.
- The command what will search a file (could be a source file, but could also be a compiled .o file or an executable) for this string and will print "the rest of the line".
- This makes it easy to check what file revisions were used to produce an executable or a .o file.

42

SCCS subdirectory

- Useful approach:
 - Your working directory (holds files to build program)
 - SCCS subdirectory

```
get SCCS/s.Comsl.cc
```
- 'make' can handle extraction of files
 - Assumes that they will be in an SCCS subdirectory

43

More sophisticated systems

- SCCS (or RCS) will work
 - For single developer wishing to maintain code in version managed system
 - For group of developers using machines with a shared file-space
 - You could do CSCI205/CSCI311/CSCI318/CSCI321 group project work using SCCS on the banshee machine
 - Get allocated a Unix group with all your members in that group
 - Use the file space of one of the group members for the SCCS archive, make sure that all files placed in archive are set to proper group
- But most cooperative developments don't have a shared file-space!
 - Think of all the open-source projects hosted at sourceforge etc.

46

SCCS - limitations

- Shared file-system only
- Access controls – Unix file-permissions by user/group
- Locking strategy
 - Often a source of problems
 - Someone checks out a file – and goes off sick
 - No-one else in the group can do anything on that file (systems admin has to login as missing user and unget the file!)
- Pretty much focussed on content files with C source code.

44

Web-access required

- Need version management systems that
 - Allow access over Internet, either
 1. Using own application protocol
 2. Tunneling via HTTP
 - Allow the definition of access controls
 - Anyone can download some files
 - Some files are restricted to particular subset of users while commits (SCCS deltas) are restricted to a smaller subsets of users
 - Restrictions involve some name/password challenge

47

Modern version control

45

CVS and Subversion

- CVS
 - Started mid 1980s
 - Client-server option
 - "Repository" (archive) on server machine
 - Clients can perform gets, "updates" (like a get – but only transfers a file if the version on the server is more recent than the copy you already have), and commits (deltas)
- Subversion
 - Started ~2000
 - Replacement for CVS
 - CVS getting so old it couldn't easily be maintained
 - Some CVS features relating to renaming of files, updates of entire directories etc had come to be regarded as limitations
- Both CVS and subversion work either in client-server mode, or can use a command-line "client" program on machine where repository is held.

48

Subversion info

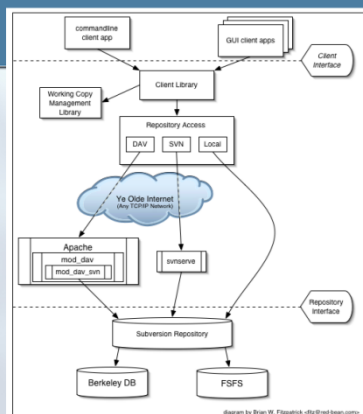
- Get latest version free on the Internet
 - Also copy in /share/cs-pub/csci222
- Lots more freebies and information at <http://subversion.tigris.org/links.html>
- How to ...?
 - Ask google

49

Numerous clients

- Subversion client integrated into development environment
- Others like:
 - TortoiseSVN
 - Extension to Window's Explorer – could be convenient if using a subversion repository for documents rather than programs
 - Subcommander – C++ GUI (!!!) using the popular Qt C++ GUI libraries from Trolltech
 - Kdesvn – Linux KDE shell
 - etc

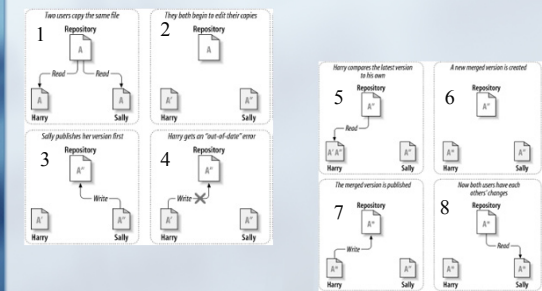
52



50

A different approach to conflicting updates

■ Copy-Modify-Merge



53

Subversion

- The repository:
 - Holds directories rather than individual files
 - Data are stored in a "database" (derived from Berkeley db system), can also have a back-up in a file system.
 - Improvements to "commit" system – now transactional (either all files get updated, or no change to repository)
- Variety of access mechanisms:
 - "Local" – use command line program **svn**
 - Svnserve – relatively simple "server" with defined access protocol, many clients
 - Apache/WebDav – access tunneling over HTTP

51

Copy-modify-merge

- Subversion's standard change process is therefore somewhat like the "three way merge" that SCCS supported.
- Like SCCS – the process cannot be fully automated:
 - When you download after hitting an "out-of-date" error on publication, you get list of differences between your version and committed version
 - You have to analyse differences looking for conflicts – places where changes overlap
 - You have to resolve conflicts by thinking about the code and re-programming as necessary.

54

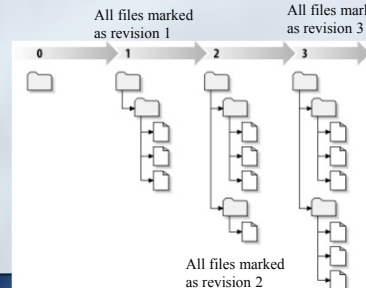
Locking

- Some files, e.g. Images, cannot really support "merging" of changes.
- Subversion does provide a lock-modify-unlock mechanism for such binary files.
- You shouldn't need to use it.

55

Repository - revisions

- Essentially, each commit creates a new directory/file tree within the repository:



58

Subversion – remember to "update"

- "Update" regularly if working on a project where others are making frequent changes
- Your copies are brought up-to-date with the versions in the repository
 - (Supposedly this is optimized, the network traffic contains only the changes to the files)

56

Repository organization: Projects: Trunk, Branches, Tags

- Typically:
 - Repository:
 - Project subdirectory-1:
 - Trunk
 - Branches
 - Tags
 - Project subdirectory-2:
 - Trunk
 - Branches
 - Tags
 - ...
 - Each "project" directory within the repository should have three subdirectories – trunk, branches, and tags

59

Revision numbers

- Subversion's handling of revision numbers is quite different from SCCS or CVS.
- Basically, a subversion "repository" represents work on a main project
 - It may define sub-projects
- When a developer commits ("publishes") changes to one or more files, this results in a new version of the overall project
- The revision numbers of all files and directories are updated (a file created at the start and never changed will still get new revision numbers when anything else changes)

57

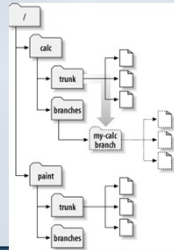
Trunk and branches

- Conceptually, trunk and branches are similar to SCCS's mainline development and branches (well, things are a bit more complex – read the subversion book!)
- As with SCCS, branches allow you to create special versions, explore different algorithms etc
 - Subversion book also suggests use of branches for large scale code reorganizations – this fits with programming styles (XP etc) where code is "refactored" regularly (less common in C++ world than Java/C#)

60

Creating a branch

- Conceptually, when you create a branch you copy an entire sub-tree from the "trunk" directory to the "branches" directory (read the subversion book – the system uses "cheap copies", relatively little data is actually duplicated)



61

Released version of software

- Your group hacks away for a while, eventually you have a version of the software for a release, if you are RUPping then you will be generating planned releases with increasing subsets of functionality
- With SCCS and similar, you would find that this release had revision 1.3 of file-1, 1.5 of file-2, 1.2 of file-3 etc etc; all a bit messy
- With subversion, your release is the current "head" of the trunk development – and all files have the same revision number.
- What do you want to do?
 - Copy that set of files to a directory where they will be preserved – not subject to any further commits.

64

Working with a branch ...

- Once a branch has been created, you can get its files, update, and commit to that branch.
- Of course, development will then be going on in parallel
 - Trunk – improvements, maybe bug fixes, extensions
 - Branch – different algorithm for X, *but I want those bug fixes as well!*

62

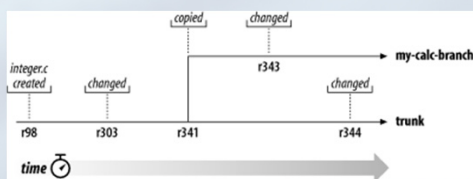
Tags

- The "tags" subdirectory is where you keep releases.
- In principle, you copy all the files from the trunk head to the tags directory
 - In practice, "cheap copying" is done so there isn't large scale duplication of files.

65

Merging changes

- Subversion provides mechanisms for merging changes made in trunk into branches
 - Cannot be fully automated, as ever there will be cases where there are conflicts between changes made in trunk code and changes already made in the branch code



63



Subversion in practice

66

First – install the subversion software somewhere!

- Versions exist for Linux and Windows.
 - Once you have the subversion software installed, you can use it much like SCCS and maintain a repository in your own local filesystem
 - You simply create a repository in some directory in your filesystem
 - The only odd thing is that subversion always requires you to use URLs – but it can be a file URL like `file:///tmp/repos/test`
- Or, on Windows, something like `file:///X:/path/to/repos` where X is the drive identifier

67

Importing your initial code

■ svn import

```
$ svn import -m"Starting" $HOME/mystuff \  
file:///tmp/svn/myrepository/trunk/mystuff
```

- svn should report on each of the files added to the repository

70

Use the command line program **svn**

■ Main svn commands

- Update your working copy
 - **svn update**
- Make changes
 - **svn add**
 - **svn delete**
 - **svn copy**
 - **svn move**
- Examine your changes
 - **svn status**
 - **svn diff**
- Undo some changes
 - **svn revert**
- Resolve Conflicts (Merge Others' Changes)
 - **svn update**
 - **svn resolved**
- Commit your changes
 - **svn commit**

68

Checkout

■ Grab the complete latest revision

```
$ svn checkout \  
file:///tmp/svn/myrepository/trunk/mystuff \  
$HOME/work
```

71

Starting – create repository

- **Read the svn book if you want to use svn on a local repository, these examples are indicative only!**

```
$ svnadmin create /tmp/svn/myrepository  
$ svn mkdir --message="Setting up the directories..." \  
file:///tmp/svn/myrepository/trunk \  
file:///tmp/svn/myrepository/tags \  
file:///tmp/svn/myrepository/branches
```

- That should create a repository for a single project
- The single project has the standard trunk, branches, tags structure

There is a tutorial on local repository based use of subversion at <http://www.onlamp.com/pub/a/onlamp/2002/10/31/subversion.html> though it might be a bit out of date

69

What do you actually get in a checkout?

- You will get the files – pretty obvious that.
- There is also a directory `.svn` (this may not appear in directory listings of course)
- This directory contains
 - All sorts of housekeeping information belonging to svn
 - Copies of the files as initially downloaded (so you get two copies, one you can see and edit, another hidden in `.svn`)
- These file copies allow svn client operations like "status" and "diff" to work even if you aren't in contact with svn repository – svn diff will compare your edited file with the copy hidden in `.svn`

72

.svn directory information

- Information in .svn directory enables "client" software to keep track of file status:
 - Unchanged, and current
 - The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its working revision. An **svn commit** of the file will do nothing, and an **svn update** of the file will do nothing.
 - Locally changed, and current
 - The file has been changed in the working directory, and no changes to that file have been committed to the repository since you last updated. There are local changes that have not been committed to the repository, thus an **svn commit** of the file will succeed in publishing your changes, and an **svn update** of the file will do nothing.
 - Unchanged, and out-of-date
 - The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the latest public revision. An **svn commit** of the file will do nothing, and an **svn update** of the file will fold the latest changes into your working copy.
 - Locally changed, and out-of-date
 - The file has been changed both in the working directory, and in the repository. An **svn commit** of the file will fail with an "out-of-date" error. The file should be updated first; an **svn update** command will attempt to merge the public changes with the local changes. If Subversion can't complete the merge in a plausible way automatically, it leaves it to the user to resolve the conflict.

73

Internet accessible repository

- Making subversion repository available to others on Internet is simply a matter of:
 - Setting up a server
 - Writing lots and lots and lots and lots of configuration and control files.

76

svn commit

- Commit changes – svn should know where to commit to (information in the .svn directory)!

`$ svn commit -m"Updated something"`

74

svnserve

- svnserve
 - Application specific protocol layered over TCP/IP
 - Windows-based
 - Server would be set up as a Windows service
 - Control-panel/Administration Tools/Services
 - Linux/Unix-based
 - Typically configured to run with inetd – when request comes to port 3690, OS will start svnserve
- Client must know and request full file path of repository
- Repository directory can contain a configuration file with names/passwords etc for users who can access files

77

But you wanted an Internet based repository ...

75

HTTP access

- Berners-Lee's original Web (>1990):
 - Read published hypertext articles
 - Navigate hypertext links
- The commercial Web (>1994)
 - Fill in a form with your order and your credit card number
 - Send (encrypted) to our server
 - We process your order, we take your money, you get something
- But weren't webs and hypertext supposed to encourage collaborative authoring?
 - In late 1990s, people interested in **Web-based Distributed Authoring and Versioning** got the W3C to authorize a project that would extend use of HTTP to support this aim.

78

WebDAV

- WebDAV – focused first (~1998) on the “distributed authoring” parts and defined a system:
 - functionality to create, change and move documents on a remote server
 - features
 - locking (overwrite prevention),
 - properties (creation, removal, and querying of information about author, modified date, etc.),
 - name space management (ability to copy and move Web pages within a server's namespace)
 - and collections (creation, removal, and listing of resources).

79

DeltaV

- DeltaV:
 - More operations added to the HTTP/WebDAV protocol
 - VERSION-CONTROL, CHECKIN, CHECKOUT, UNCHECKOUT, REPORT
 - LABEL (add names to revisions)
 - UPDATE.
 - MERGE.
 - MKACTIVITY
 - MKWORKSPACE
 - BASELINE-CONTROL

82

<http://www.webdav.org/deltav/WWW10/deltav-intro.htm>

WebDAV

- Builds on HTTP
 - Works in terms of “resources” (defined URL)
 - These are typically “collections” – directory hierarchy with the files created by the distributed authors
 - uses XML documents for properties etc
- Defines a set of operations:
 - PROPFIND —
 - Retrieve properties from a resource.
 - Also download a directory hierarchy from a remote system.
 - PROPPATCH —
 - Change properties
 - MKCOL
 - Create collections
 - COPY/MOVE
 - Operate on collections
 - LOCK/UNLOCK

80

HTTP/WebDAV/DeltaV protocols

DeltaV Web Versioning and Configuration Management Protocol
CHECKIN, CHECKOUT, UNCHECKOUT, VERSION-CONTROL, REPORT, UPDATE, LABEL, MERGE, MKWORKSPACE, BASELINE-CONTROL, MKACTIVITY
WebDAV Distributed Authoring Protocol (RFC 2518)
LOCK, UNLOCK, PROPFIND, PROPPATCH, COPY, MOVE, MKCOL
HyperText Transfer Protocol (HTTP) 1.1 (RFC 2616, RFC 2617)
GET, HEAD, POST, PUT, DELETE, OPTIONS, TRACE, CONNECT

83

WebDAV

- WebDav HTTP extensions supported by some operating systems and browsers
 - E.g. Microsoft created a shared file system mechanism that used WebDav protocols.
- WebDAV
 - “Distributed Authoring”
 - But no versioning!

81

WebDAV/DeltaV

- All a bit clunky.
- But it is a standard protocol – not something that belongs to Microsoft.
- So anyone can create a WebDAV/DeltaV aware client.
- On the server –
 - It is just an issue of an HTTP server being confronted with ~18 additional commands that supplement the standard HTTP protocol's GET/POST/PUT/OPTIONS etc

84

Apache – the “patchy” server

- Apache can be extended:
 - A core set of modules – a simple HTTP server that returns static files (HTML, text, Images)
 - Add modules
 - Mod_auth: Apache can deal with name/password access controls
 - Mod_cgi: Apache knows CGI and can launch processes
 - Mod_access: Apache can restrict access by IP address
 - Mod_php: Apache incorporates a PHP interpreter
 - So, how to handle WebDAV/DeltaV – write a new module for Apache

85

Configured for multiple repositories

- A repository is basically a single project (with optional subprojects) with trunk/branch/tag elements; revision numbers apply to entire repository.
- Typically, your Apache server will need to support independent projects for different developer groups.
- Each project then needs its own repository.
- Repositories need to be set up before any attempt is made to access them remotely
 - Svn commands to create repository, add subprojects if needed, add trunk/branch/tag directories to each subproject
 - Register a file (if just a README) – otherwise you get an error when client invocation touches an empty database!
 - Set up access controls

88

Apache modules

- Two modules
 - mod_dav: Apache can intercept the WebDAV/DeltaV requests
 - mod_dav_svn: These requests are converted into “subversion” repository actions.
- Apache configuration file extended with directives
 - Use the mod_dav_svn module for any requests involving “location svn”
 - Find the svn repository at this file path ...

86

Access control – small extension to normal Apache access control

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn-repos
  AuthType Basic
  AuthName "Subversion Repository"
  AuthUserFile /etc/apache2/dav_svn.passwd
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>

LimitExcept: here allow anyone to do a “Get” or “Report”

But you will need to be a valid-user to do a commit.
```

89

Apache setup

- Apache 2 (modern threaded version)
 - Is still a standard web-server, serving HTML documents etc from wherever its HTDOCS directory has been configured
 - Has this extra ability to operate on subversion repositories in the file-space of its host machine
- Has its own authentication mechanisms
 - Own user/password file (independent of any users/passwords defined for its Linux or Windows host OS)

87

Access controls

- Apache supplies user-password and group files.
- A particular repository can be restricted to
 - Any valid-user (anyone who can enter a name/password combination found in the Apache file)
 - A specified subset of named users
 - Or any member of a group

90



Handle with care ...

- There is no systems administrator tending this WebDAV server (only an academic)
- If you crash the system while doing a commit you may "Wedge" your database – making it unusable.
- Tough.
- I don't know the magic invocations needed to repair databases.

94

Setup

- Apache
 - Approximately 25 repositories created
 - Apache user-name passwords
 - For each group, 5 members Group1M1 etc
 - Each repository restricted to members of a group with no "Limit-Except" (so no general access)
 - Each repository created with two projects each having the normal trunk/branch/tag structure and a README file in the trunk

92



Use

- Your only access will be through HTTP/WebDAV clients – such as the subversion client added to NetBeans
- All group members should be able to:
 - Add files
 - Checkout and update files
 - Commit files

93

GIT

- A completely ignorant, childish person with no manners.
- A person who feels justified in their callow behaviour.
- A pubescent kid who thinks it's totally cool to act like a moron on the internet, only because no one can actually reach through the screen and punch their lights out
- Total and utter tosser who is incapable of doing anything other than annoying people, and not in a way that is funny to others.

(Urban dictionary)

96

GIT

- Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.
- Torvalds has quipped about the name git, which is British English slang roughly equivalent to "unpleasant person". Torvalds said: *"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."*
- The UK IT jobs website itjobswatch.co.uk reports that as of February 2014, approximately 16.6% of UK permanent software development job openings list Git as a requirement, compared to 16.9% for Subversion, 11.4% for Microsoft Team Foundation Server, 1.0% for Visual SourceSafe, and 1.95% for Mercurial.

97

GIT

- Git encourages you to have multiple local branches that can be entirely independent of each other. The creation, merging, and deletion of those lines of development takes seconds. So why not try: <http://git-scm.com/>
 - **Frictionless Context Switching.**
Create a branch to try out an idea, commit a few times, switch back to where you branched from, apply a patch, switch back to where you are experimenting, and merge it in.
 - **Role-Based Codelines.**
Have a branch that always contains only what goes to production, another that you merge work into for testing, and several smaller ones for day to day work.
 - **Feature Based Workflow.**
Create new branches for each new feature you're working on so you can seamlessly switch back and forth between them, then delete each branch when that feature gets merged into your main line.
 - **Disposable Experimentation.**
Create a branch to experiment in, realize it's not going to work, and just delete it - abandoning the work—with nobody else ever seeing it (even if you've pushed other branches in the meantime).

GIT – originally for Linux

- Background: Linux development
 - Multiple developers spread across the world
 - For several years, used a proprietary source code management (SCM) system BitKeeper
 - Break in 2005 with owner of that software
- Linus Torvalds decides to create new SCM designed specifically for needs of the community of Linux developers
 - First version took him about 3 days to create!
- Why is it growing?
 - **Better support for branching**
 - Suits developments like Linux where have numerous contributors working independently
 - Encourages experimentation even in smaller projects
 - If combining different versions is made easier, it's more tempting to create a branch to test a "mad" idea

98

GIT – Overview (from Wikipedia)

- Distributed development
 - Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

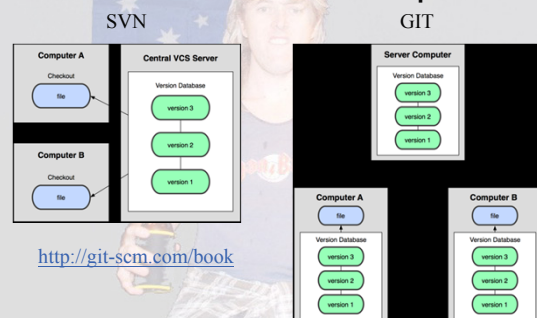
101

GIT – Overview (from Wikipedia)

- Strong support for non-linear development
 - Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers. Branches in git are very lightweight:
 - A branch in git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

99

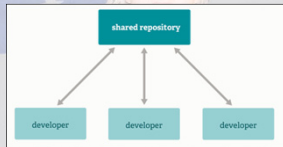
GIT – distributed development



With GIT, you get a complete copy to work on (so if you need to do something like look at an earlier version of a file there is no network latency and the operation is much quicker)

102

GIT – distributed development



Your work group can combine their individual efforts into a central repository – i.e. work in a manner similar to using subversion.

But other strategies are supported that are better suited to larger scale and more distributed development groups ...

103

GIT – Overview (from Wikipedia)

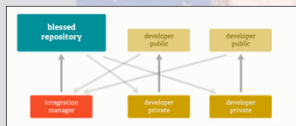
■ Efficient handling of large projects

- Performance tests done by Mozilla showed GIT was an order of magnitude faster than some version-control systems, and fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server

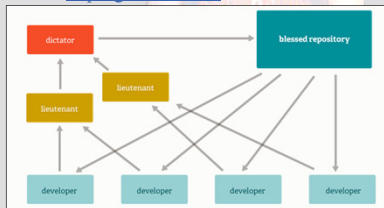


106

GIT – distributed development



<http://git-scm.com/>



Model for most open-source GitHub projects.
Developers add their changes to public directories
Project czar adds from these to master copy

Linux scale –
Linux has lieutenants that look after particular aspects of Linux (e.g. filesystem)

104

GIT – Overview (from Wikipedia)

■ Cryptographic authentication of history

- The Git history is stored in such a way that the ID of a particular version (a commit in Git terms) depends upon the complete development history leading up to that commit.

■ Toolkit-based design

- Git was designed as a set of programs written in C, and a number of shell scripts that provide wrappers around those programs
 - Although most of those scripts have since been rewritten in C for speed and portability, the design remains, and it is easy to chain the components together

107

GIT – Overview (from Wikipedia)

■ Compatibility with existing systems/protocols

- Repositories can be published via HTTP, FTP, rsync, or a Git protocol over either a plain socket, or ssh.
 - Git server can support old CVS clients
 - Git server can work with SVN repositories

105

GIT – Overview (from Wikipedia)

■ Pluggable merge strategies

- Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and that manual editing is required.

■ Garbage accumulates unless collected

- Aborting operations or backing out changes will leave useless dangling objects in the database.
Git will automatically perform garbage collection when enough loose objects have been created in the repository.

108

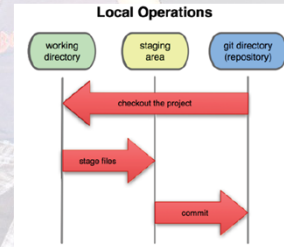
GIT – Overview (from Wikipedia)

- Periodic explicit object packing
 - A new object added – a new file
 - Uses of lots of files takes space and has efficiency issues
 - Periodically (end of work day?), any new objects get moved into "packfiles"
 - Packfiles are structured with an index that has the offsets of the individual objects

109

GIT – staging area

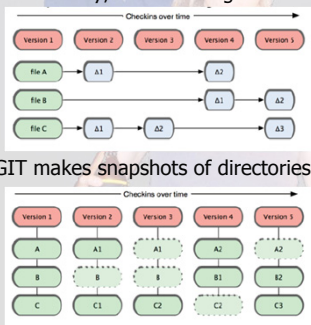
- You select the modified files that will be included in the next commit that you do
 - So you can have many changed files; some that you think are ready to commit, others you want to hold back as you are still working on them



112

GIT – "filesystems" rather than files

- Traditionally, record changes in individual files:
 - SCCS deltas – the changes in each file
- GIT makes snapshots of directories
 - (avoids duplication of unchanged files)



110

GIT

- Download and install the software for your system
 - <http://git-scm.com/downloads>
- It's basically a command line program (*it was after all created for Linux gurus!*)
 - Of course there are GUI clients,
 - There is a client built into NetBeans
- You can get the code for an existing project from a repository, e.g. github,
 - `git clone git://github.com/adobe/brackets`
- Or, you can start a new git repository in the directory that holds your own project
 - `git init`

113

GIT – *one place where it's more expensive ...*

- Somewhat expensive to get the change history of an individual file!
- GIT must walk the global history and determine for each snapshot whether there was a change in the file

111

GIT at the command line

- Read the book!
 - <http://git-scm.com/book>
- `git add *.c`
- etc

114

GIT or SVN

- Back in 2007 when I created the practical component for CSCI222, the obvious choice for a version management system was SVN
- If I was making the choice in 2014?

Probably GIT

115